Korespondenční Seminář z Programování

38. ročník **KSP** **Únor 2026**

Vzorová řešení třetí série třicátého osmého ročníku KSP

38-3-1 Zahrada s bonsajemi

V úlohe máme rozrezať strom na niekoľko podstromov, tak, že každý vrchol bude práve v jednom z výsledných stromov, všetky výsledné stromy budú mať priemer najvýše Z a odstránime pritom čo najmenší počet hrán. Zamyslime sa nad tým, ako budú vyzerať výsledné stromy – potom prídeme na to, ako ich vyrobiť. Konkrétne spravme pár pozorovaním ktoré musia platiť pre korene týchto stromov a hrany z nich:

- Koreň pôvodného stromu (vrchol 1) bude vždy koreňom nejakého výsledného stromu.
- Pokiaľ mala nejaká hrana idúca z koreňa dĺžku väčšiu než Z, určite sme ju odrezali.
- Pokiaľ mala nejaká hrana z koreňa dĺžku Z a z koreňa išla aspoň jedna ďalšia hrana, jednu z nich sme museli odrezať – a určite sa nám oplatilo odrezať tú dlhšiu (ak by koreň mal viac než dve hrany, opačný prístup by problém nevyriešil).
- Pokiaľ mala nejaká hrana dĺžku $Z-1$ a existovala aspoň jedna hrana s dĺžkou väčšou než 1, oplatilo sa nám odrezať tú dlhšiu (z rovnakého dôvodu).

Skúsme to ďalej zovšeobecniť:

- Pokiaľ existovala vo vrchole dvojica hrán dĺžky l_i a $Z - l_i + 1$, dlhšiu z nich sme určite odrezali.

Toto tvrdenie ale nie je pravdivé – predstavme si takýto „strom“ s koreňom vo vrchole a a limit $Z = 5$:

(e) − − (d) − − − (a) − − (b) − − − − (c)

Odrezanie hrany (a, d) by viedlo k stromom s koreňmi a a d, ale strom a by bol príliš veľký a bolo by treba odrezať aj hranu (b, c) (alebo (a, b)). Pritom odrezaním hrany (a, b) vzniknú dva správne veľké stromy, a teda je to jediné optimálne riešenie. Zrejme sa teda nestačí pozerať na jednotlivé dĺžky hrán, ale musíme sa dívať na celé dĺžky ich podstromov. Tie ale môžeme ovplyniť tým, že ich samotné rozrežeme na menšie časti.

Pridajme ešte jedno pozorovanie:

- Keďže nás zaujíma čo najmenší počet podstromov, určite sa neoplatí odrezať nejaký podstrom skôr než to bude nevyhnutné – teda pokiaľ máme dve hrany (so spoločným vrcholom, pričom jedna smeruje k rodičovi daného vrcholu), ktoré by sme mohli rozrezať tak, že obidva rezy by delili aktuálny (príliš veľký) podstrom na dva zenové (rozumne malé) podstromy, vždy sa oplatí použiť tú, ktorá je bližšie ku koreňu. Podstrom ďalej od koreňa bude stále zenový, ale ten bližšie bude najviac rovnako dlhý a možno dokonca kratší, než v opačnom prípade – a to nám môže pomôcť s delením podstromov ďalej smerom ku koreňu.

To nás vedie k *pažravému* (hladovému, greedy) rekurzívnemu riešeniu aplikovanému od listov ku koreňu – najskôr vyriešime minimálne počty rezov pre podstromy, tie budú mať po optimálnom narezaní nejakú dĺžku (pokiaľ je viac možností, chceme vždy vrátiť tú najmenšiu možnú). Dĺžky podstromu pridáme k dĺžkam hrán, ktoré do nich vedú, a dostaneme *efektívne* dĺžky hrán. Potom postupne odrezávame efektívne najdlhšie hrany tak, aby nezostali žiadne dve, ktoré majú v súčte dĺžku viac než Z (a samozrejme žiadna, ktorá je sama dlhšia než Z). Pri odrezaní hrany prehlásime jej koreň za kořeň nového podstromu.

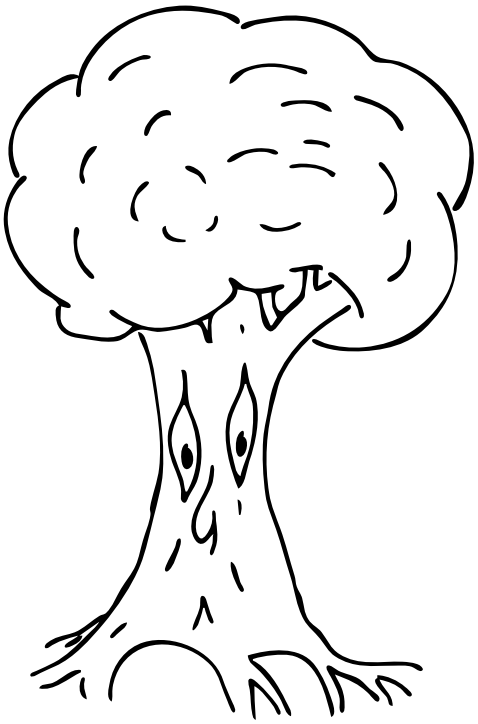
Pokiaľ si hrany v každom vrchole najskôr spočítame a potom utriedime podľa ich efektívnych dĺžok, stačí nám na zistenie kolidujúcich dvojíc prejsť postupnosť raz, pamätať si najdlhšiu hranu, ktorú sme v strome ešte nechali, a ostatné postupne odrezávať. Toto riešenie má časovú zložitosť $O(N \log N)$, kvôli triedeniu vo vrcholoch.

Program (Python):
`http://ksp.mff.cuni.cz/viz/38-3-1.py`

Úloha sa však dá vyriešiť aj s lineárnou časovou zložitosťou – uvedomme si, že *krátkych* hrán s efektívnou dĺžkou $\leq Z/2$ môžeme mať v podstrome ľubovoľne veľa, ale *dlhú* hranu s väčšou dĺžkou najviac jednu (a aj to iba v prípade, že sa zmestí do podstromu naraz s najdlhšou *krátkou* hranou). Takže nám stačí pamätať si najdlhšiu krátku hranu, ktorú sme v podstrome zatiaľ videli, a najkratšiu dlhú hranu, a môžeme hrany prechádzať v ľubovoľnom poradí. Pokiaľ narazíme na dlhú hranu, a už nejakú inú dlhú hranu v podstrome máme, odrežeme dlhšiu z nich a kratšiu si necháme. Pokiaľ narazíme na krátku hranu, môže sa stať, že sa nezmestí spolu s dlhou hranou, ktorú si pamätáme – potom dlhú hranu proste odrežeme a nebudeme mať žiadnu dlhú hranu (ak nenájdeme ďalšiu, ktorá sa do stromu zmestí).

Program (Python):
`http://ksp.mff.cuni.cz/viz/38-3-1-linear.py`

Úlohu připravili: David Klement, Ján „Janči“ Plachý



38-3-2 Stavba sjezdovky

Jako první si sestavíme sjezdovku. Chceme pro každé místo na sjezdovce v konstantním čase vědět, jestli se na něm nachází překážka. Toho docílíme například 2D polem o obou rozměrech daných délkou sjezdovky.

Teď nás zajímá největší trojúhelník na sjezdovce, který neobsahuje překážku. Budeme se snažit stavět trojúhelníky z menších trojúhelníků. Hlavní pro nás bude koukat se na jejich vrchní vrcholy (říkejme jim vrchy). Všimneme si vlastnosti, že pokud jsou dva vrcholy vedle sebe vrchy dvou prázdných trojúhelníků stejné výšky, pak pokud je vrchol nad nimi prázdný, tak je vrchem o 1 delšího prázdného trojúhelníku. To bude hlavní myšlenka našeho algoritmu.

Postupovat budeme inkrementálně:

Pomalé řešení

Budeme postupně zvyšovat velikost prázdných trojúhelníků, které nás budou zajímat. Vrchy trojúhelníků o délce 1 už známe, to je prakticky naše sestrojená sjezdovka.

Poté si vytvoříme další prázdnou kopii sjezdovky, kam budeme evidovat vrchy prázdných trojúhelníků s o 1 vyšší délkou. Pro každý volný vrchol na sjezdovce se stačí podívat, jestli jsou dva vrcholy pod ním vrchy trojúhelníků z předchozí fáze.

Takhle postupujeme, dokud máme nějaký vrch dané velikosti. Pokud nemáme, víme že už větší prázdný trojúhelník nebude, tedy vezmeme nějaký vrch z předchozí fáze a máme řešení.

Teď si ještě všimněme, že nepotřebujeme více než dvě poslední fáze kopií sjezdovky, čímž zredukujeme potřebný prostor na velikost sjezdovky. Ve skutečnosti nám dokonce bude stačit pouze jedna, pokud se na ni budeme dívat odspodu a pod rukama si ji přepisovat.

Toto řešení nám dává paměťovou složitost v $\mathcal{O}(N^2)$ – velikost sjezdovky, časovou v $O(N^3)$ – fází může být až N a v každé v každém vrcholu sjezdovky vykonáme konstantní práci.

Lineární řešení

Budeme se držet myšlenky, že se na vrcholy sjezdovky koukáme odspodu. Zkusíme k algoritmu dojít systematicky – na čem trávíme spoustu zbytečného času?

Tak zaprvé, vrcholy nemohou být vrchy trojúhelníků s délkou větší, než je jejich výška. Pojďme se ale na zbytečné trávení času podívat i z druhé strany – na začátku zjišťujeme, jestli jsou vrcholy nahoře vrchy trojúhelníků o délce 1, 2, 3, ... To ale vlastně nepotřebujeme, pokud během vyhodnocování daného vrcholu již známe vyšší délku prázdného trojúhelníku pro dva vrcholy pod ním. Tedy ve chvíli, kdy víme, zda jsou dva nižší vrcholy vrchy trojúhelníků o délce i, dokážeme rovnou zjistit, zda je vrchol nad nimi trojúhelníkem o délce $i + 1$ a nemusíme se zdržovat předchozími fázemi.

Jediná potíž je nakonec v tom, že musíme pro ty dva nižší vrcholy vědět, jakého největšího trojúhelníku jsou vrchem. Všimněme si, že to ale závisí pouze na vrcholech, které jsou níž. Takže postupujeme od spodu.

Základ indukce

Vytvoříme si sjezdovku, kde si pamatujeme *největší délku prázdného trojúhelníku, jehož je každý vrchol vrchem*. Té budeme říkat *hledaná délka* pro daný vrchol. Nejprve v nejnižší výšce zjistíme, zda jsou jeho vrcholy prázdné – tedy kdy mají hledanou délku 1. Tím pro ně zjistíme hledanou délku – ta nemůže být vyšší než 1.

Správnost dokážeme skrz invariant, že v každém vrcholu je to, co si uložíme, opravdu jeho hledanou délkou. Pro první fázi to tedy platí. Postupujme induktivně – ukažme, že pokud to platí pro předchozí výšku, platí to i pro tu hned nad ní.

Indukční krok

Pokračujme dál: pro každý vrchol v dané výšce nejprve zjistíme, jestli je prázdný – pokud není, můžeme rovnou říct, že není vrchem žádného trojúhelníku, tedy je jeho hledaná délka nulová a invariant platí.

Pokud je ale prázdný, podíváme se na délku dvou vrcholů pod ním, což už máme spočítané. Víme, že hledaná délka vyššího vrcholu nemůže být větší, než délka jednoho vrcholu ze dvou pod ním plus jedna. Kdyby byla, byla by hledaná délka nějakého vrcholu pod ním vyšší, než co jsme si uložili, což je spor s naším invariantem z indukčního předpokladu, a tedy se to nemohlo stát.

Teď se nám stačí podívat na minimum z nich dvou. Víme, že oba vrcholy pod aktuálním jsou vrchy prázdných trojúhelníků s danou délkou. Také víme, že pro alespoň jeden z nich je to hledaná délka. Z toho plyne, že pokud k jeho hledané délce přičteme jedničku, je to hledaná délka pro vrchol výš – vlastně to je akorát aplikované pozorování ze začátku řešení, s přidanou hodnotou toho, že je to maximální taková délka pro jednoho ze synů. Invariant tedy i v této situaci platí.

Algoritmus

Tím jsme indukcí poměrně složitě dokázali něco docela jednoduchého. Zformulujeme tedy algoritmus:

1. $S \leftarrow$ Sjezdovka
2. $V \leftarrow S$ (Zde budou hledané délky)
3. Pro h od N do 1:
4. Pro v od 1 do h:
5. Pokud $S[h][i] = 0$:
6. $V \leftarrow 0$
7. Jinak:
8. Pokud $h = 1$:
9. $V[h][i] \leftarrow 1$
10. Jinak:
11. $V[h][i] \leftarrow \min(V[h-1][i], V[h-1][i-1]) + 1$

Tím pro každý vrchol zjistíme jeho hledanou délku. Jako výsledek prohlásíme maximální číslo, které někam uložíme, společně s jeho pozicí – to si můžeme pamatovat průběžně.

Jakou to tedy má složitost? Na každý vrchol se podíváme jednou pro vyhodnocení jeho hledané délky a až dvakrát kvůli vrcholům těsně nad ním. Tedy je časová složitost v $\mathcal{O}(N^2)$. Paměť potřebujeme pouze na dvě sjezdovky – jedna pro vstup a jedna pro výsledky – tedy v $\mathcal{O}(N^2)$. (Což také můžeme zkomprimovat do jedné. :))

Program (Python):
`http://ksp.mff.cuni.cz/viz/38-3-2.py`

Erikovo řešení

Toto je řešení Erika Ježka, které pro vás sepsal. Jelikož je lepší než naše autorské, udělili jsme mu zaň bonusové body. Díky Eriku!

Pro lepší představu trojúhelník (sjezdovku) můžeme zarovnat doleva (tedy všechna i-tá políčka řádků jsou nad sebou). Představme si, že jsme v nějakém řádku r, pak pro každé políčko si spočítáme, v jakém řádku se nachází první překážka směrem nahoru - označme tyto hodnoty $A_r[i]$.

Pokud máme mít trojúhelník, který má základnu na řádku r, začíná na pozici l a má délku strany x, tak musí platit $A_r[l] \leq r - x, A_r[l+1] \leq r - x + 1, A_r[l+2] \leq r - x + 2, \ldots$ To je zatím taková nemastná, neslaná podmínka, ale vidíme, že je ekvivalentní s $\max(A_r[l], A_r[l+1]-1, A_r[l+2]-2, \ldots) \leq r - x$. To je výrazně lepší a mělo by dost navádět na *trik odečtení přímky*, tedy definujeme $B_r[l] = A_r[l] - l$. Podmínka přejde na $\max(B_r[l], B_r[l+1], ...) \leq r - x - l$.

Všimněme si, že jsme nikdy neřekli, až po kterou hodnotu $B_r[l+y]$ bereme maximum, protože to není potřeba. Je snadné si rozmyslet, že můžeme brát celý suffix.

Takže podmínka říká, že pro fixní l nás zajímá suffixové maximum z B_r. Pro každý vrchol tedy umíme ze suffixového maxima získat největší možnou velikost prázdného podtrojúhelníku začínajícího v něm. Budeme si udržovat pouze úsečky, kde prvky jedné mají stejné suffixové maximum.

Taková struktura bude vypadat tak, že poslední úsečka má nějakou hodnotu, úsečka před ní o něco vyšší, a tak dále – neboli maxima úseček rostou z prava do leva.

Zbývá rozmyslet, jak strukturu udržovat při přechodu na další řádek, neboli změnu $B_r \to B_{r+1}$. Naštěstí se hodnoty $B_r[i]$ téměř nezmění, přibude nám jedna nová na konci a některé se zvětší v důsledku překážky na novém řádku. Díky tomu, že se nikdy nesníží, je jednoduché si udržovat úsečky v setu. Když přijde překážka, najdeme úsečku, kam patří. Pokud se zvýšilo maximum pro daný bod, zvýšilo se i u levé strany úsečky a u všech úseček nalevo s nižším maximem – ty všechny sloučíme do jedné.

Ještě zbývá objasnit, že překážky potřebujeme setřídit, abychom je mohli přidávat postupně – primárně podle vyšky, sekundárně podle umístění ve výšce.

Označme N = délka strany sjezdovky a P = počet překážek. Se setem interagujeme za každé z N přidaných polí a za každou z P přidaných překážek. Jaká je složitost operací se setem? Úseček může být maximálně N, takže se nabízí vzít operaci v $\mathcal{O}(\log N)$. Ale při $P \approx N^2$ je $\log P \approx 2 \log N$, jenže při $P \ll N$ si použitím P jako velikosti setu pomůžeme, tedy odhadneme operaci na $O(\log P)$. Případné procházení úseček nalevo se naúčtuje jejich vytvoření. Třídit tedy můžeme libovolně v $\mathcal{O}(P \log P)$ a celková složitost je stále v $\mathcal{O}((N+P)\log P)$.

Nakonec si pojďme ukázat, jak řešení ještě zlepšit. Přidaná políčka na koncích řádků nemusíme přidávat rovnou, ale až když je potřebujeme. Neboli pokud na řádku nepřidáváme překážku, tak ho vůbec neřešíme (ano, budeme *líní*). Pak před přidáním překážky můžeme ještě přidat úsečku, která shrne všechna přeskočená přidaná pole na koncích řádků, páč je jejich společné maximum 0. Tím si ušetříme N samostatných operací se setem a vždy je shrneme až při nějaké z P operací. Tím máme časovou složitost v $\mathcal{O}(P \log P)$.

Úlohu připravili: Jan Adámek, Patrik Prítrský, Matúš Púll, Jirka Sejkora

38-3-3 Těžká úloha

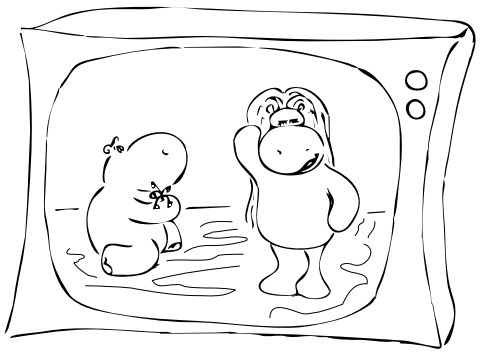
Připomeňme si nejprve, že máme za úkol dokázat, že je Kevinův problém NP-těžký. Jinými slovy chceme dokázat, že lze každý problém ze třídy NP převést v polynomiálním čase na tento problém.

Převodem problému A na problém B v polynomiálním čase myslíme, že navrhneme polynomiální algoritmus, který přeloží libovolný vstup problému A na vstup problému B tak, že z řešení problému B jiným algoritmem v polynomiálním čase získáme řešení problému A.

Naštěstí nemusíme pracně vymýšlet univerzální postup, který by takto dokázal každý existující problém ze třídy NP převést na Kevinův problém. Můžete si rozmyslet, že pokud umíme převést v polynomiálním čase problém A na problém B a problém B na problém C, umíme v polynomiálním čase převést i A na C.

Bohatě nám tedy bude stačit vzít libovolný problém, o kterém již někdo dokázal, že lze na něj převést každý problém ze třídy NP, a převést ho v polynomiálním čase na Kevinův problém.

Hledanou vlastnost mají například NP-úplné problémy, kterých je v naší kuchařce uvedeno hned několik. My si z nich vybereme problém Součet podmnožiny.



Stavba vhodného grafu

V problému Součet podmnožiny máme zadaný seznam nezáporných celých čísel a speciální číslo k. Cílem je zjistit, zda existuje podmnožina čísel, jejíž součet je přesně k.

Zkusme si rozmyslet, jak převést libovolný vstup tohoto problému na vstup Kevinova problému. Kevinův problém je grafový, takže by mohlo pomoci postavit takový graf, ve kterém bude libovolná cesta ze startu do cíle reprezentovat nějakou konkrétní volbu podmnožiny čísel.

Abychom si usnadnili práci, budeme hledat místo grafu vhodný multigraf. To nám nijak nevadí, protože lze jakýkoliv multigraf převést na graf tím, že každou jeho paralelní hranu rozdělíme novým pomocným vrcholem na dvě hrany, kde jedné hraně dáme délku a cenu původní hrany, a druhé nastavíme obě hodnoty na 0.

Uvažme multigraf, ve kterém se pro N čísel nachází $N+1$ vrcholů, a má podobný tvar jako orientovaná cesta, akorát mezi sousedními dvěma vrcholy vede místo jedné hrany dvojice různě ohodnocených hran.

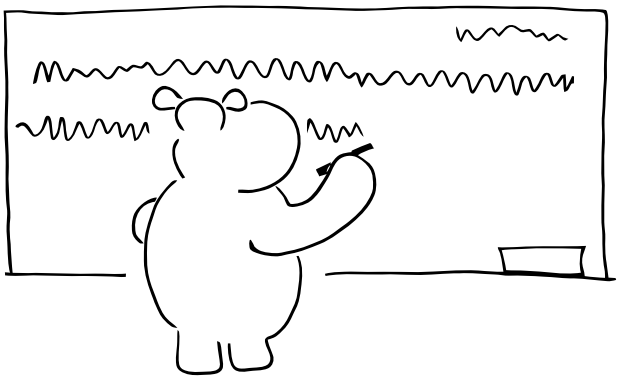
Vrchol, do kterého nevstupuje žádná hrana, prohlásíme za start a vrchol, ze kterého žádná hrana nevystupuje, za cíl. Každá dvojice hran bude přiřazena jednomu konkrétnímu číslu ze seznamu, a bude reprezentovat naši volbu, zda chceme do podmnožiny zahrnout toto číslo či nikoliv.

Nechť je c_i hodnota i-tého čísla. Ceny a délky potom určíme pro i-tou dvojici hran následovně:

První hrana bude odpovídat volbě vložit číslo i do podmnožiny. Délka hrany bude c_i a její cena 0.

Druhá hrana bude naopak odpovídat volbě ponechat předmět i mimo podmnožinu a její délka bude 0 a její cena c_i.

Můžeme si ověřit, že nyní každá cesta ze startu do cíle jednoznačně odpovídá nějaké podmnožině a naopak. Navíc se celková délka každé cesty rovná součtu čísel ležících v odpovídající podmnožině a celková cena cesty se rovná součtu čísel, která se v dané podmnožině nenacházejí.



Dokončení převodu

Nalezli jsme potřebný graf, pojďme vymyslet, na jakou hodnotu nastavit K, tedy maximální cenu, kterou je Kevin ochoten utratit, a ukázat, jak potom z nejkratší cesty splňující omezení zjistit, zda existuje podmnožina se součtem k.

Nechť je S součet všech čísel ze seznamu. Ukážeme, že je správná volba nastavit K na hodnotu $S - k$.

Nejkratší cesta splňující omezení bude vždy existovat, protože se dá ze startu do cíle dostat jen po hranách s nulovou cenou.

Dále si můžeme rozmyslet, že celková délka nejkratší cesty nikdy nebude menší než k. Pokud by totiž byla ostře menší než k, byl by součet nezahrnutých čísel větší než $S - k$ a cesta by proto nesplnila omezení na cenu.

Nastavením maximální ceny na $S - k$ jsme tedy vynutili, že má nejkratší cesta splňující omezení vždy délku alespoň k. Protože zároveň víme, že každá cesta odpovídá jednoznačně nějaké podmnožině, existuje v tomto grafu nejkratší cesta délky k, právě když existuje podmnožina součtu k, a v opačném případě je nejkratší cesta delší než k.

Máme tedy funkční převod. Můžeme si rozmyslet, že pro vstup o velikosti N je graf velký $\mathcal{O}(N)$, takže je převod skutečně polynomiální.

Další převody

I další problémy z kuchařky se dají přímočaře převést na Kevinův problém. Pojďme se na ně na závěr ve stručnosti podívat:

Začněme převodem problému Dva loupežníci. Ten je totožný jako v případě Součtu podmnožiny, jen se místo libovolného k pracuje s k rovnajícím se polovině součtu všech čísel.

Zajímavější je převod problému Batohu. Připomínáme, že v rozhodovací verzi z kuchařky nás zajímá, jestli je možné vybrat takové předměty, jejichž celková váha nepřesáhne kapacitu batohu, a jejich celková cena je alespoň k.

I v tomto převodu budeme pro každý předmět vybírat z dvojice hran, z nichž jedna bude reprezentovat vložení předmětu do batohu, a druhá ponechání předmětu mimo batoh. Graf tedy bude mít úplně stejný tvar jako v případě Součtu podmnožiny, jen je opět třeba vhodně zvolit ceny a váhy hran.

Zvolme si číslo M, které odpovídá ceně nejdražšího předmětu. Cenu i-tého předmětu budeme značit c_i. Hrana, která odpovídá vložení předmětu i do batohu, bude mít cenu rovnající se váze předmětu i. Délku hrany nastavíme na $M - c_i$. Druhá hrana bude mít nulovou cenu a délku M. Hodnotu K nastavíme na kapacitu batohu.

Můžeme si rozmyslet, že každá cesta opět jednoznačně odpovídá umístění předmětů do batohu a naopak. Navíc ceny hran spolu s K vynucují, že součet vah předmětů nepřekročí kapacitu batohu.

Každá cesta nyní bude mít délku $N \cdot M - S$, kde S je součet cen předmětů vybraných touto cestou. Kevinův problém se snaží minimalizovat délku cesty, a protože je $N \cdot M$ konstantní hodnota pro všechny cesty, dosáhne toho nalezením maximálního možného součtu cen předmětů nepřesahujících kapacitu batohu.

Na závěr stačí zkontrolovat, jestli je maximální součet cen větší nebo rovný k, a podle toho odpovědět.

Úlohu připravil: David Kolář

38-3-4 Medvědi

Úlohu budeme řešit tak, jak jsme naznačili v zadání – nejprve postavíme vhodnou tokovou síť a následně v ní nalezneme maximální tok. Na základě maximálního toku potom rozhodneme, zda je možné zachránit všechny účastníky, a pokud ano, získáme z něj i posloupnost instrukcí pro každého z nich.

Pojďme si představit základní princip naší sítě:

Síť se bude skládat z $S + 1$ vrstev, kde vrstva i bude vhodným způsobem reprezentovat mapu bezpečných políček lesa na konci taktu i, tedy mapu těch políček, která nejsou na konci taktu i v dosahu nějakého medvěda. Vrstvu 0 budeme chápat, jako vrstvu mapující stav lesa před začátkem všech pohybů.

Mezi vrstvami natáhneme orientované hrany tak, aby platilo, že mezi reprezentací políčka a ve vrstvě i a reprezentací políčka b ve vrstvě $i+1$ existuje orientovaná hrana, právě když je možné, aby se účastník stojící na konci taktu i v lese na bezpečném políčku a přesunul během taktu $i+1$ na bezpečné políčko b jedním ze základních pohybů nahoru, dolů, doleva, doprava či zůstat stát na místě.

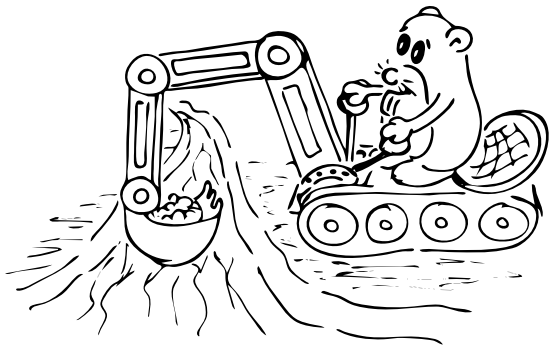
Můžeme si rozmyslet, že každá cesta v takové síti odpovídá jednoznačně nějaké bezpečné posloupnosti pohybů účastníka v lese, začínající a končící v konkrétním čase (na konci taktu) na konkrétní pozici. To samé platí i naopak.

Každá správná síť musí mít také svůj zdroj a stok. Zdroj budeme chápat jako místo, ze kterého účastníci přišli, a bude propojen hranami s těmi políčky z nulté vrstvy, na kterých na počátku účastníci stojí. Stok budeme zase interpretovat jako pomyslné útočiště, kam se musí účastníci dostat, aby přežili.

Stok bude spojen hranami jednak s políčky z poslední vrstvy, čímž budeme reprezentovat situaci, kdy šli medvědi spát a účastníci jsou již v bezpečí, a zároveň s krajními políčky, což zase bude odpovídat opuštění lesa skrz jeho okraj.

Protože se lze dívat na pohyb v lese každého přeživšího účastníka jako na nějakou konkrétní cestu skrz tuto síť zdroje do stoku, a cesty účastníků (mimo zdroj a stok) musí být kvůli tomu, že smí na každém políčku stát nejvýše jeden účastník, nutně disjunktní, hledáme vlastně U disjunktních cest mezi zdrojem a stokem v této síti, což je, jak víte z kuchařky, problém řešitelný pomocí problému maximálního toku.

Ve výsledné síti bude skutečně platit, že půjde zachránit všech U účastníků, právě když bude existovat U disjunktních cest, a tedy maximální tok se v naší síti bude rovnat U.



Stavba sítě podrobněji

V předchozí sekci jsme nebyli příliš konkrétní, pojďme proto popsat, jak přesně bude naše síť vypadat a jak bude velká.

Víme, že se ve vrstvě i budou nacházet jen políčka, která jsou na konci taktu i bezpečná. Nebudeme políčka nicméně reprezentovat jako jeden uzel, ale pomocí dvou uzlů - vstupního a výstupního.

Jejich role je zřejmá, do vstupního uzlu budou vstupovat hrany z předchozí vrstvy a z výstupního uzlu budou naopak vystupovat hrany do vrstvy následující. Z vstupního do výstupního uzlu povede orientovaná hrana kapacity 1. Tímto rozdělením uzlu jsme vynutili, že může na odpovídajícím políčku na konci taktu i stát nejvýše jeden účastník.

Z výstupního uzlu políčka povedou orientované hrany kapacity 1 do vstupních vrcholů těch políček, kam se dá dostat jedním ze základních 5 pohybů.

Může se nicméně stát, že se políčko nachází na kraji lesa a nějaký ze 4 sousedů mu chybí. V takovém případě z něj natáhneme orientovanou hranu kapacity 1 do stoku.

Do stoku také povedou orientované hrany kapacity 1 z každého výstupního uzlu nacházejícího se v poslední vrstvě, tedy ve vrstvě S.

Ze zdroje povedou orientované hrany kapacity 1 do vstupních uzlů těch políček, na kterých se nachází účastníci na počátku.

Můžeme si nyní rozmyslet, že je tato síť acyklická, a tak má každý tok v této síti velikosti k tvar k vrcholově disjunktních cest, což odpovídá tomu, že odstraníme ty účastníky, do jejichž výchozích pozic nevedou ze zdroje hrany s tokem 1, a zbylým dáme odpovídající instrukce k přežití podle jejich příslušné cesty.

To samé se dá udělat i naopak, a ze sady instrukcí pro nějakou podmnožinu účastníků můžeme vytvořit odpovídající tok.

Z toho plyne, že má maximální tok velikost U, právě když existuje sada instrukcí, která zachrání všechny účastníky. Díky pozorování výše můžeme tuto instrukci pohybů snadno z toku vyčíst.

Pojďme se ještě kvůli analýze časové složitosti podívat na velikost výsledné sítě. V každé z S vrstev se nachází nejvýše P políček, takže je počet uzlů této sítě $\mathcal{O}(SP)$. Zároveň pro každý uzel, který není zdroj, platí, že z něj vychází nejvýše 5 hran, takže je nezdrojových hran $\mathcal{O}(SP)$. Protože ze zdroje vychází $\mathcal{O}(P)$ hran, je celkový počet hran sítě konstantakrát větší než počet uzlů, a tedy $\mathcal{O}(SP)$.

Hledání maximálního toku

Protože každý medvěd ohrožuje jen konstantní počet políček, dá se snadno postavit výsledná síť v čase $\mathcal{O}(SP)$, a naše řešení tak bude záviset jen na vhodném algoritmu, který použijeme pro hledání maximálního toku.

Pojďme si proto na závěr pro zajímavost říct něco o algoritmech, které se dají k nalezení maximálního toku použít, a vybrat z nich nějaký vhodný.

Nechť je m počet hran sítě a n počet vrcholů. Naší první volbou by mohl být Ford-Fulkerson z kuchařky používající prohledávání do šířky. Ten běží v obecné tokové síti v čase $\mathcal{O}(m^2 \cdot n)$.

V Průvodci labyrintem algoritmů si můžete přečíst o tom, že se dá na jeho základu vybudovat trochu složitější, ale rychlejší Dinicův algoritmus. Jeho přímočará implementace má v obecné tokové síti časovou složitost $\mathcal{O}(m \cdot n^2)$.

Pokud má nicméně síť (tak jako ta naše) asymptoticky stejně hran, jako vrcholů, není tento algoritmus v obecné síti rychlejší, než byl Ford-Fulkerson.

Mohli bychom proto sáhnout po nějaké sofistikovanější variantě Dinicova algoritmu. Existuje například verze využívající pokročilou datovou strukturu zvanou Link/cut stromy. Ta běží v obecné síti v čase $\mathcal{O}(mn \cdot \log n)$, což by v síti, kde je počet hran asymptoticky stejně velký, jako počet vrcholů, bylo skutečně rychlejší.

My si nicméně místo pokročilých triků všimneme toho, že naše toková síť rozhodně není obecná toková síť a má poměrně specifické vlastnosti.

Například má jednotkovou kapacitu hran. Pro takovou síť platí, že v ní obyčejný Dinicův algoritmus z průvodce doběhne v čase $\mathcal{O}(m^{3/2})$. Pokud vás zajímá důkaz, můžete nahlédnout do skript Krajinou grafových algoritmů, kde se nachází i další zajímavosti týkající se Dinicova algoritmu.

S Dinicovým algoritmem máme řešení v čase $\mathcal{O}((SP)^{3/2})$.

Dodejme ještě, že špatně by si v této síti nevedl ani Ford-Fulkerson, který se z počátku jevil jako příliš pomalý. Platí, že najde zlepšující cestu v čase lineárním k velikosti sítě, tedy v $\mathcal{O}(SP)$. Protože zároveň víme, že je maximální tok nejvýše tak velký, jako je počet účastníků U, nalezne zlepšující cestu nejvýše U krát.

Protože U náleží $\mathcal{O}(P)$, získáme odhad složitosti $\mathcal{O}(SP^2)$. Pokud by U byla konstanta (což jsme ale v zadání rozhodně netvrdili), tak by tento algoritmus dokonce běžel v lineárním čase vzhledem k velikosti sítě.

Úlohu připravil: David Kolář

38-3-S Kouzlo hešování

Úkol 1: Worst-case zvětšování tabulky

Když nám dojde místo v tabulce velikosti N, chceme si pořídit novou tabulku velikosti $2N$ (s novou hešovací funkcí) a přesunout do ní všechny prvky ze staré tabulky. To obnáší novou tabulku vynulovat, projít všechny přihrádky staré tabulky, z každé vysbírat prvky a jeden po druhém je vkládat do tabulky nové. To dohromady trvá $\Theta(N)$ času. Abychom zajistili worst-case konstantní složitost *Insert*u, musíme celou reorganizaci tabulky rozprostřít přes následujících N *Insert*ů. Na každý *Insert* tedy zbude konstantně mnoho času.

Především to tedy znamená, že během reorganizace bude existovat současně stará i nová tabulka. Nové *Insert*y budou vkládat do nové tabulky, *Find* se musí dívat do obou. Po N *Insert*ech bude reorganizace hotová a současně nová tabulka velikosti $2N$ akorát zaplněna (N prvků ze staré tabulky a N nových), tím pádem můžeme navázat další reorganizací.

Tento přístup má ovšem jeden zásadní háček (přímo hák): už v prvním *Insert*u, který nastane během reorganizace, potřebujeme vkládat do nové tabulky. Jenže tu jsme ještě nestihli celou vynulovat. Musíme na to proto jít o něco chytřeji: reorganizaci spustíme, když bude stará tabulka plná ze 3/4. Během následujících $N/4$ *Insert*ů budeme ještě vkládat do staré tabulky, ale během každého *Insert*u vynulujeme 8 přihrádek nové tabulky. Tím pádem v okamžiku zaplnění staré tabulky bude nová připravena.

Teď se ale začnou reorganizace překrývat: ještě než je reorganizace hotova, je už nová tabulka ze 3/4 zaplněna, takže bychom měli spustit další reorganizaci. Raději velikost nové tabulky nastavíme na $3N$ místo $2N$, takže 3/4 její velikosti bude $3/4 \cdot 3N = 9/4 \cdot N > 2N$. Tím zajistíme, že se další reorganizace spustí až po ukončení té předchozí. Jen to ještě způsobí, že během nulování tabulky budeme muset inicializovat 12 přihrádek v jednom kroku místo 8.

Algoritmus popíšeme podrobně. N bude značit velikost staré tabulky (BÚNO násobek čtyř), $3N$ velikost nové a n aktuální počet prvků v množině.

Insert(x):

1. $n \leftarrow n + 1$
2. Jestliže $n < 3/4 \cdot N$:
3. Vložíme x do staré tabulky.
4. Jinak je-li $n < N$: ◁ *nulování nové tabulky*
5. Pokud $n = 3/4 \cdot N$:
6. Alokujeme novou tabulku velikosti $3N$.
7. $p \leftarrow 0$ ◁ *pozice během nulování*
8. Vynulujeme přihrádky p až $p + 11$ v nové tabulce.
9. $p \leftarrow p + 12$
10. Vložíme x do staré tabulky.
11. Jinak je-li $n < 2N$: ◁ *přesun prvků ze staré do nové*
12. Pokud $n = N$: ◁ *začneme procházet starou tabulku*
13. $i \leftarrow 0$ ◁ *ve které přihrádce jsme*
14. Opakujeme dvakrát:
15. Pokud *stará*[i] není prázdná:
16. Odebereme první prvek x ze *stará*[i].
17. Vložíme x do nové tabulky.
18. Jinak: ◁ *přechod na další přihrádku*
19. $i \leftarrow i + 1$
20. Jinak: ◁ $n = 2N$, *reorganizace hotova*
21. Starou tabulku zrušíme.
22. Nová se stane starou.
23. $N \leftarrow 3N$

Proč v 14. kroku opakujeme dvakrát: celkem potřebujeme N-krát vyzvednout prvek z přihrádky a N-krát se posunout na další přihrádku. To je celkem $2N$ kroků, které rozkládáme přes N *Insert*ů.

Úkol 2: Kolize funkce f

Zkusíme náhodně generovat 64-bitové prvky a počítat jejich heše tak dlouho, než se nějaký heš zopakuje. K tomu si budeme v nějaké slovníkové datové struktuře (třeba hešovací tabulce :)) pamatovat, které heše jsme už viděli a z jakých prvků jsme je získali.

Jak rychlé to bude? Jelikož hešujeme do prostoru velikosti $m = 2^{32}$, podle narozeninového paradoxu se dají očekávat první kolize po řádově $\sqrt{m} = 2^{16} = 65\,536$ náhodných pokusech. To je příjemně málo.

Obecně by algoritmus pro b-bitové heše měl průměrnou časovou složitost $2^{b/2}$.

Úkol 3: Multikolize funkce f

Nejprve ukážeme heuristický, nicméně docela dobře funkční přístup. Pořídíme si číslo z_0 takové, že $f(z_0) = 0$. Budeme ho hledat náhodným zkoušením; jelikož $f(z_0)$ nabývá 2^{32} možných hodnot, do hledaného z_0 se strefíme s pravděpodobností $1/2^{32}$. Podle lemmatu o džbánu tedy uspějeme za průměrně 2^{32} pokusů.

Podívejme se dovnitř funkce f. Aby po `>> 32` vyšel výsledek 0, musí výsledek násobení být tvaru `00000000xxxxxxxx` hexadecimálně, kde `x` jsou nějaké neznámé číslice.

Doufali bychom, že $2z_0$ se vynásobí na `00000000yyyyyyyy`, tím pádem $f(2z_0)$ vyjde opět 0. Jenže pokud `xxxxxxxx` bylo větší nebo rovno `80000000`, dolních 8 číslic přeteče do těch horních a vyjde `00000001yyyyyyyy` a $f(2z_0) = 1$.

Pomůžeme si jednoduchým trikem: najdeme z_{-1}, pro nějž $f(z_{-1}) = 2^{32} - 1$. Tím pádem násobení uvnitř výpočtu $f(z_{-1})$ dává `ffffffffwwwwwwww` pro neznámé `w`.

Nyní pokud $2z_0$ během násobení přeteče, použijeme místo toho $2z_0 + z_{-1}$, které nejspíš přeteče zpět a opět vyjde v horních 8 číslicích 0. Pokud by se náhodou stalo, že v horních číslicích budou samá `f`, opravíme to přičtením z_0. Tyto dvě úpravy budeme střídat, dokud v horních 8 číslicích nevyjdou samé nuly.

Obecněji to bude vypadat takto:

1. $x \leftarrow 0$
2. Dokud nemáme dost kolizí:
3. $\quad x \leftarrow x + z_0$
4. $\quad$ Je-li $h(x) = 0$:
5. $\quad\quad$ Vypíšeme x.
6. $\quad$ Jinak je-li $h(x) < 2^{31}$:
7. $\quad\quad x \leftarrow x + z_{-1}$
8. $\quad$ Jinak:
9. $\quad\quad x \leftarrow x + z_0$

Jak rychlé to bude pro b-bitovou hešovací funkci? Počáteční hledání z_0 a z_{-1} trvá průměrně 2^b kroků. Kolik přičtení z_0 a odečtení z_{-1} v průměru potřebujeme na jednu kolizi, neumíme spočítat přesně, ale kdyby se h chovala náhodně, bylo by to v průměru konstantně (opět díky lemmatu o džbánu).

Jak pomůže teorie čísel

Místo lehce pochybné randomizované heuristiky můžeme kolizi hledat naprosto deterministicky mašinerií teorie čísel (viz kuchařka).

Podívejme se, co dělá funkce $f(x)$. Nejprve spočítá $\psi(x) = (ax) \bmod 2^b$ a z tohoto čísla následně vykousne horních $b/2$ bitů (pro nějaké hodnoty a a b). Do přihrádky 0 se tedy zahešují právě ta x, pro která je $\psi(x) \in \{0, \ldots, 2^{b/2} - 1\}$.

Rovnice $\psi(x) = t$ je ovšem lineární kongruence, která podle kuchařky má právě jedno řešení, kdykoliv je a nesoudělné s 2^b. To je v našem případě splněno, jelikož a je liché. Řešení najdeme rozšířeným Euklidovým algoritmem v čase $\mathcal{O}(b)$.

Pro každé $t \in \{0, \ldots, 2^{b/2} - 1\}$ tedy umíme v čase $\mathcal{O}(b)$ najít jedno x, pro nějž je $f(x) = 0$. Navíc všechna tato x jsou navzájem různá. Nalezení k násobné kolize tedy potrvá $\mathcal{O}(kb)$.

Úkol 4: Kolize funkce g

Kolidující dvojici hezkých řetězců opět najdeme náhodným zkoušením. Budeme vytvářet náhodné 6-znakové řetězce z velkých písmen, malých písmen a číslic. Těch je $(26 + 26 + 10)^6 = 62^6 \doteq 56 \cdot 10^9$, což je bohatě přes $2^{32} \doteq 4 \cdot 10^9$. Hešováním těchto řetězců bychom tedy měli dostávat přibližně rovnoměrně náhodné výsledky (to nebudeme pořádně dokazovat a vlastně to ani nepotřebujeme, jelikož nerovnoměrnost by nám jenom pomohla – viz poznámka u narozeninového paradoxu v zadání).

Hešujeme do prostoru velikosti 2^{32}, takže po průměrně 2^{16} pokusech najdeme kolidující dvojici.

V obecném případě s b-bitovým výsledkem, zvolíme řetězce délky $\ell = \lceil b / \log_2 52 \rceil$ (aby bylo $2^\ell \geq 2^b$). Potřebujeme jich vyzkoušet řádově $2^{b/2}$, každý pokus trvá $\mathcal{O}(\ell)$ na vygenerování řetězce a výpočet funkce g a průměrně $\mathcal{O}(\ell)$ na hledání ve slovníku známých hešů. Celkem tedy algoritmus poběží v čase $\mathcal{O}(2^{b/2} \cdot \ell) = \mathcal{O}(2^{b/2} \cdot b)$.

Úkol 5: Multikolize funkce g

Všimneme si, že $g(x_0 x_1 \ldots x_{\ell-1})$ vlastně vyhodnocuje polynom s koeficienty x_0 až $x_{\ell-1}$:

$$\left(a^\ell + \sum_{i=0}^{\ell-1} x_i \cdot a^{\ell-1-i} \right) \bmod 2^b,$$

kde a a b jsou nějaké konstanty.

Z toho plyne, že slepíme-li za sebe řetězce $x_0 \ldots x_{\ell-1}$ a $y_0 \ldots y_{\ell-1}$, výsledek se zahešuje na

$$\left(a^{2\ell} + \sum_{i=0}^{\ell-1} x_i \cdot a^{2\ell-1-i} + \sum_{i=0}^{\ell-1} y_i \cdot a^{\ell-1-i} \right) \bmod 2^b =$$
$$= \left(a^\ell \cdot g(x_0 \ldots x_{\ell-1}) + g(y_0 \ldots y_{\ell-1}) - a^\ell \right) \bmod 2^b.$$

Tím pádem pokud řetězec x koliduje s nějakým x' a y s nějakým y', tak všechny řetězce xy, xy', $x'y$ a $x'y'$ také kolidují.

Takže najdeme-li kolidující pár x a x', všechny řetězce slepené z t částí, z nichž každá je x nebo x', budou navzájem kolidovat. Tím získáme 2^t kolidujících řetězců.

Jak je to rychlé? Pro obecné b a k-násobnou kolizi nejprve v čase $\mathcal{O}(2^{b/2} \cdot b)$ najdeme jeden kolidující pár (x, x') délky $\ell \approx b$. Pak budeme vytvářet všechny řetězce složené z $t = \lceil \log_2 k \rceil$ kopií x nebo x'. To zvládneme v čase $\mathcal{O}(2^t \cdot t \cdot \ell) = \mathcal{O}(k \cdot \log k \cdot b)$.

Program (C++):
`http://ksp.mff.cuni.cz/viz/38-3-s-kolize.cpp`

Úlohu připravil: Martin „Medvěd“ Mareš

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*